

SAMPLE CHAPTER

Covers Spring 5.0

# Spring IN ACTION

FIFTH EDITION

Craig Walls



 MANNING



*Spring in Action*  
*Fifth Edition*

by Craig Walls

**Chapter 19**

Copyright 2019 Manning Publications

# *brief contents*

---

<b>PART 1</b>	<b>FOUNDATIONAL SPRING .....</b>	<b>1</b>
	1 ■ Getting started with Spring	3
	2 ■ Developing web applications	29
	3 ■ Working with data	56
	4 ■ Securing Spring	84
	5 ■ Working with configuration properties	114
<b>PART 2</b>	<b>INTEGRATED SPRING .....</b>	<b>135</b>
	6 ■ Creating REST services	137
	7 ■ Consuming REST services	169
	8 ■ Sending messages asynchronously	178
	9 ■ Integrating Spring	209
<b>PART 3</b>	<b>REACTIVE SPRING .....</b>	<b>239</b>
	10 ■ Introducing Reactor	241
	11 ■ Developing reactive APIs	269
	12 ■ Persisting data reactively	296

<b>PART 4</b>	<b>CLOUD-NATIVE SPRING.....</b>	<b>321</b>
13	■ Discovering services	323
14	■ Managing configuration	343
15	■ Handling failure and latency	376
<b>PART 5</b>	<b>DEPLOYED SPRING .....</b>	<b>393</b>
16	■ Working with Spring Boot Actuator	395
17	■ Administering Spring	429
18	■ Monitoring Spring with JMX	446
19	■ Deploying Spring	454

# 19

## *Deploying Spring*

---

### ***This chapter covers***

- Building Spring applications as either WAR or JAR files
- Pushing Spring applications to Cloud Foundry
- Containerizing Spring applications with Docker

Think of your favorite action movie. Now imagine going to see that movie in the theater and being taken on a thrilling audiovisual ride with high-speed chases, explosions, and battles, only to have it come to a sudden halt before the good guys take down the bad guys. Instead of seeing the movie's conflict resolved, when the theater lights come on, everyone is ushered out the door. Although the lead-up was exciting, it's the climax of the movie that's important. Without it, it's action for action's sake.

Now imagine developing applications and putting a lot of effort and creativity into solving the business problem, but then never deploying the application for others to use and enjoy. Sure, most applications we write don't involve car chases or explosions (at least I hope not), but there's a certain rush you get along the way. Not every line of code you write is destined for production, but it'd be a big let-down if none of it ever was deployed.

Up to this point, we've focused on using the features of Spring Boot that help us develop an application. There have been some exciting steps along the way. But it's all for nothing if you don't cross the finish line and deploy the application.

In this chapter, we're going to step beyond developing applications with Spring Boot and look at how to deploy those applications. Although this may seem obvious for anyone who has ever deployed a Java-based application, there are some unique features of Spring Boot and related Spring projects you can draw on that make deploying Spring Boot applications unique.

In fact, unlike most Java web applications, which are typically deployed to an application server as WAR files, Spring Boot offers several deployment options. Before we look at how to deploy a Spring Boot application, let's consider all the options and choose a few that suit your needs best.

## 19.1 Weighing deployment options

You can build and run Spring Boot applications in several ways. The appendix covers many of them, including these:

- Running the application in the IDE with either Spring Tool Suite or IntelliJ IDEA
- Running the application from the command line using the Maven `spring-boot:run` goal or Gradle `bootRun` task
- Using Maven or Gradle to produce an executable JAR file that can be run at the command line or deployed in the cloud
- Using Maven or Gradle to produce a WAR file that can be deployed to a traditional Java application server

Any of these choices is suitable for running the application while you're still developing it. But what about when you're ready to deploy the application into a production or other non-development environment?

Although running an application from the IDE or via Maven or Gradle aren't considered production-ready options, executable JAR files and traditional Java WAR files are certainly valid options for deploying applications to a production environment. Given the options of deploying a WAR file or a JAR file, how do you choose? In general, the choice comes down to whether you plan to deploy your application to a traditional Java application server or to a cloud platform:

- *Deploying to Java application servers*—If you must deploy your application to Tomcat, WebSphere, WebLogic, or any other traditional Java application server, you really have no choice but to build your application as a WAR file.
- *Deploying to the cloud*—If you're planning to deploy your application to the cloud, whether it be Cloud Foundry, Amazon Web Services (AWS), Azure, Google Cloud Platform, or most any other cloud platform, then an executable JAR file is the best choice. Even if the cloud platform supports WAR deployment, the JAR file format is much simpler than the WAR format, which is designed for application server deployment.

In this chapter, we'll focus on three deployment scenarios:

- Deploying a Spring Boot application as a WAR file to a Java application server such as Tomcat
- Pushing a Spring Boot application as an executable JAR file to Cloud Foundry
- Packaging a Spring Boot application into a Docker container for deployment to any platform that supports Docker deployments

To get started, let's take a look at how you can build the ingredient service application into a WAR file that can be deployed to a Java application server such as Tomcat.

## 19.2 *Building and deploying WAR files*

Throughout the course of this book, as you've developed the applications that make up the Taco Cloud application, you've run them either in the IDE or from the command line as an executable JAR file. In either case, an embedded Tomcat server (or Netty, in the case of Spring WebFlux applications) has always been there to serve requests to the application.

Thanks in large part to Spring Boot autoconfiguration, you've been spared from having to create a `web.xml` file or servlet initializer class to declare Spring's `DispatcherServlet` for Spring MVC. But if you're going to deploy the application to a Java application server, you're going to need to build a WAR file. And, so that the application server will know how to run the application, you'll also need to include a servlet initializer in that WAR file to play the part of a `web.xml` file and declare `DispatcherServlet`.

As it turns out, building a Spring Boot application into a WAR file isn't all that difficult. In fact, if you chose the WAR option when creating the application through the Initializr, then there's nothing more you need to do.

The Initializr ensures that the generated project will contain a servlet initializer class and the build file will be geared to produce a WAR file. If, however, you chose to build a JAR file from the Initializr (or if you're curious as to what the pertinent differences are), then read on.

First, you'll need a way to configure Spring's `DispatcherServlet`. Whereas this could be done with a `web.xml` file, Spring Boot makes this even easier with `SpringBootServletInitializer`. `SpringBootServletInitializer` is a special Spring Boot-aware implementation of Spring's `WebApplicationInitializer`. Aside from configuring Spring's `DispatcherServlet`, `SpringBootServletInitializer` also looks for any beans in the Spring application context that are of type `Filter`, `Servlet`, or `ServletContextInitializer` and binds them to the servlet container.

To use `SpringBootServletInitializer`, create a subclass and override the `configure()` method to specify the Spring configuration class. Listing 19.1 shows `IngredientServiceServletInitializer`, a subclass of `SpringBootServletInitializer` that you'll use for the ingredient service application.

**Listing 19.1 Enabling Spring web applications via Java**

```
package tacos.ingredients;

import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.context.web.SpringBootServletInitializer;

public class IngredientServiceServletInitializer
    extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(
        SpringApplicationBuilder builder) {
        return builder.sources(IngredientServiceApplication.class);
    }
}
```

As you can see, the `configure()` method is given a `SpringApplicationBuilder` as a parameter and returns it as a result. In between, it calls the `sources()` method that registers Spring configuration classes. In this case, it only registers the `IngredientServiceApplication` class, which serves the dual purpose of a bootstrap class (for executable JARs) and a Spring configuration class.

Even though the ingredient service application has other Spring configuration classes, it's not necessary to register them all with the `sources()` method. The `IngredientServiceApplication` class, annotated with `@SpringBootApplication`, implicitly enables component scanning. Component scanning discovers and pulls in any other configuration classes that it finds.

For the most part, `SpringBootServletInitializer`'s subclass is boilerplate. It references the application main configuration class. But aside from that, it'll be the same for every application where you'll be building a WAR file. And you'll almost never need to make any changes to it.

Now that you've written a servlet initializer class, you must make a few small changes to the project build. If you're building with Maven, the change required is as simple as ensuring that the `<packaging>` element in `pom.xml` is set to `war`:

```
<packaging>war</packaging>
```

The changes required for a Gradle build are similarly straightforward. You must apply the `war` plugin in the `build.gradle` file:

```
apply plugin: 'war'
```

Now you're ready to build the application. With Maven, you'll use the Maven wrapper script that the Initializr used to execute the package goal:

```
$ mvnw package
```

If the build is successful, then the WAR file can be found in the target directory. On the other hand, if you were using Gradle to build the project, you'd use the Gradle wrapper to execute the build task:

```
$ gradlew build
```

Once the build completes, the WAR file will be in the build/libs directory. All that's left is to deploy the application. The deployment procedure varies across application servers, so consult the documentation for your application server's specific deployment procedure.

It may be interesting to note that although you've built a WAR file suitable for deployment to any Servlet 3.0 (or higher) servlet container, the WAR file can still be executed at the command line as if it were an executable JAR file:

```
$ java -jar target/ingredient-service-0.0.19-SNAPSHOT.war
```

In effect, you get two deployment options out of a single deployment artifact!

#### **MICROSERVICES IN APPLICATION SERVERS?**

The ingredient service application is intended to be one of several applications that are microservice constituents of the larger Taco Cloud application. But here, we're talking about deploying the ingredient service as a standalone application to an application server. Does that even make sense?

Microservices are generally like any other application and should be deployable on their own. Although the ingredient service may not be useful outside the context of the rest of the Taco Cloud application, there's no reason you can't deploy it to Tomcat or another application server. But don't expect the same ability to scale the application individually as you would get if deploying it to the cloud.

Although WAR files have been the workhorses of Java deployment for over 20 years, they were truly designed for deploying applications to a traditional Java application server. Depending on the platform you choose, modern cloud deployment doesn't require WAR files and some may not even support them. As we move into a new era of cloud deployment, perhaps JAR files are a better choice.

### **19.3 *Pushing JAR files to Cloud Foundry***

Server hardware can be expensive to purchase and maintain. Properly scaling servers to handle heavy loads can be tricky and even prohibitive for some organizations. These days, deploying applications to the cloud is a compelling and cost-effective alternative to running your own data center.

Several cloud choices are available, but those that offer a platform as a service (PaaS) are among the most compelling. PaaS offers a ready-made application deployment platform with several add-on services (such as databases and message brokers) to bind to your applications. In addition, as your application requires additional

horsepower, cloud platforms make it easy to scale up (or down) your application on the fly by adding and removing instances.

Cloud Foundry is an open source PaaS platform that originated at Pivotal, the same company that sponsors the Spring Framework and the other libraries in the Spring platform. One of the most compelling things about Cloud Foundry is that it offers both open source and commercial-based distributions, giving you the choice of how and where you use Cloud Foundry. It can even be run inside the firewall in a corporate data center, offering a private cloud.

Whereas Cloud Foundry will be happy to accept WAR files, the WAR file format is overkill for Cloud Foundry's needs. A simpler executable JAR file is a more suitable choice for deploying to Cloud Foundry.

To demonstrate how to build and deploy an executable JAR file to Cloud Foundry, you're going to build the ingredient service application and deploy it to Pivotal Web Services (PWS), a public Cloud Foundry hosted by Pivotal at <http://run.pivotal.io>. If you want to work with PWS, you'll need to sign up for an account. PWS offers \$87 of free trial credit and doesn't even require you to give any credit card information during the trial.

Once you've signed up for PWS, you'll need to download and install the `cf` command-line tool from <https://console.run.pivotal.io/tools>. You'll use the `cf` tool to push applications to Cloud Foundry. But the first thing you'll use it for is to log into your PWS account:

```
$ cf login -a https://api.run.pivotal.io
API endpoint: https://api.run.pivotal.io

Email> {your email}

Password> {your password}

Authenticating...
OK
```

Great! Now you're ready to take the ingredient service to the cloud! As it turns out, the project is ready to be deployed to Cloud Foundry. All you need to do is build it and then push it to the cloud.

To build the project with Maven, you can use the Maven wrapper to execute the package goal (you'll find the resulting JAR file in the target directory):

```
$ mvnw package
```

With Gradle, you can use the Gradle wrapper to execute the build task (you'll find the resulting JAR file in the build/libs directory):

```
$ gradlew build
```

Now all that's left is to push the JAR file to Cloud Foundry using the `cf` command:

```
$ cf push ingredient-service -p target/ingredient-service-0.0.19-SNAPSHOT.jar
```

The first argument to `cf push` is the name given to the application in Cloud Foundry. In this case, the full URL for the application will be <http://ingredient-service.cfapps.io>. Among other things, this name will be used as the subdomain where the application is hosted. Therefore, it's important that the name you give the application be unique so that it doesn't collide with any other applications deployed in Cloud Foundry (including those deployed by other Cloud Foundry users).

Because dreaming up a unique name can be tricky, the `cf push` command offers a `--random-route` option that randomly produces a subdomain for you. Here's how to push the ingredient service application to generate a random route:

```
$ cf push ingredient-service \
  -p target/ingredient-service-0.0.19-SNAPSHOT.jar \
  --random-route
```

When using `--random-route`, the application name is still required, but two randomly chosen words will be appended to it to produce the subdomain.

Assuming everything goes well, the application should be deployed and ready to handle requests. Supposing that the subdomain is `ingredient-service`, you can point your browser to <http://ingredient-service.cfapps.io/ingredients> to see it in action. You should receive, as a response, a list of available ingredients.

As written, the application will continue to use the embedded Mongo database (which is only intended for testing purposes) to hold ingredient data. You'll likely want to use a real database in production. At the time I'm writing this, there's a fully managed MongoDB service available in PWS under the name `mlab`. You can find this service (and any other available services) by using the `cf marketplace` command. To create an instance of the `mlab` service, use the `cf create-service` command:

```
$ cf create-service mlab sandbox ingredientdb
```

This creates an `mlab` service with the `sandbox` service plan named `ingredientdb`. Once the service is created, you can bind it to your application with the `cf bind-service` command. For example, to bind the `ingredientdb` service to the `ingredient` service application, use this:

```
$ cf bind-service ingredient-service ingredientdb
```

Binding a service to an application merely provides the application with details on how to connect to the service with an environment variable named `VCAP_SERVICES`. It doesn't change the application in any way to use that service. Once the service is bound, you'll need to re-stage the application to have the binding take effect:

```
$ cf restage ingredient-service
```

The `cf restage` command forces Cloud Foundry to redeploy the application and reevaluate the `VCAP_SERVICES` value. When it does, it'll see that there's a MongoDB

service bound to the application. It uses that service as the backing database for the application.

There are dozens of available services in PWS that you can bind your application to, including MySQL databases, PostgreSQL databases, and even ready-to-use Eureka and Config Server services. I encourage you to read more about what PWS has to offer at <https://console.run.pivotal.io/marketplace> and acquaint yourself with how to use PWS by reading the documentation at <https://docs.run.pivotal.io/>.

Cloud Foundry is a great PaaS for Spring Boot application deployment. Its association with the Spring projects affords some synergy between the two. But another common way to deploy applications in the cloud, especially when pushing to an Infrastructure-as-a-Service (IAAS) platform like AWS, is to package the application within a Docker container that's published to the cloud. Let's see how to create a Docker container that carries your Spring Boot application.

## 19.4 Running Spring Boot in a Docker container

Docker (<https://www.docker.com/>) has become the de facto standard for distributing applications of all kinds for deployment in the cloud. Many different cloud environments, including AWS, Microsoft Azure, Google Cloud Platform, and Pivotal Web Services (to name a few) accept Docker containers for deploying applications.

The idea of containerized applications, such as those created with Docker, draws analogies from real-world intermodal containers. With regard to shipping items, intermodal containers all have a standard size and format, regardless of their contents. Because of that, intermodal containers are easily stacked on ships, carried on trains, or pulled by trucks. In a similar way, containerized applications share a common container format that can be deployed and run anywhere, regardless of the application inside.

Although creating Docker images isn't terribly difficult, Spotify has created a Maven plugin that makes creating a Docker container from the result of a Spring Boot build as easy as whistling your favorite tune. To use the Docker plugin, add it to your Spring Boot project pom.xml file under the <build>/<plugins> block as follows:

```
<build>
  <plugins>
  ...
    <plugin>
      <groupId>com.spotify</groupId>
      <artifactId>dockerfile-maven-plugin</artifactId>
      <version>1.4.3</version>
      <configuration>
        <repository>
          ${docker.image.prefix}/${project.artifactId}
        </repository>
        <buildArgs>
          <JAR_FILE>target/${project.build.finalName}.jar</JAR_FILE>
        </buildArgs>
      </configuration>
    </plugin>
  </plugins>
</build>
```

```

    </configuration>
  </plugin>
</plugins>
</build>

```

Under the `<configuration>` block, you'll set a few properties to guide the creation of the Docker image. The `<repository>` element describes the name of the Docker image as it'll appear in a Docker repository. As specified here, the name is based on the Maven project artifact ID, prefixed with a value resolved from the Maven property named `docker.image.prefix`. Although the project artifact ID is something Maven already knows, you'll need to specify the prefix property:

```

<properties>
  ...
  <docker.image.prefix>tacocloud</docker.image.prefix>
</properties>

```

If this were the Taco Cloud ingredient service, the resulting Docker image would reside in the Docker repository as `tacocloud/ingredient-service`.

Under the `<buildArgs>` element, you can guide the image to include the JAR file that the Maven build produces. As shown, it uses the Maven property `project.build.finalName` to determine the name of the JAR file that's in the `target` directory.

Aside from the information you provided in the Maven build specification, all Docker images are defined from a file named `Dockerfile`. This file identifies an image to base the new image on, environment variables that should be set, any volumes that should be mounted, and (most importantly) the *entry point*—a command to execute when a container based on the image starts. For the purposes of most any Spring Boot application, the following `Dockerfile` is a great way to begin:

```

FROM openjdk:8-jdk-alpine
ENV SPRING_PROFILES_ACTIVE docker
VOLUME /tmp
ARG JAR_FILE
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java", \
            "-Djava.security.egd=file:/dev/./urandom", \
            "-jar", \
            "/app.jar"]

```

Breaking this Docker file down line by line, you see the following:

- The `FROM` instruction identifies an image to base the new image on. The new image extends the base image. In this case, the base image is `openjdk:8-jdk-alpine`, a container image based on version 8 of OpenJDK.
- The `ENV` instruction sets an environment variable. You're going to override a few of the Spring Boot application configuration properties based on the active profile, so in this image, you'll set the environment variable `SPRING_PROFILES_ACTIVE`

to `docker` to ensure that the Spring Boot application starts with `docker` as the active profile.

- The `VOLUME` instruction creates a mount point in the container. In this case, it creates a mount point at `/tmp` so that the container can write data, if necessary, to the `/tmp` directory.
- The `ARG` instruction declares an argument that can be passed in at build time. In this case, it declares an argument named `JAR_FILE`, which is the same as the argument given in the Maven plugin's `<buildArgs>` block.
- The `COPY` instruction copies a file from a given path to another path. In this case, it copies the JAR file specified in the Maven plugin to a file named `app.jar` within the image.
- The `ENTRYPOINT` instruction describes what should happen when the container starts. Given as an array, it specifies the command line to execute. In this case, it uses the `java` command line to run the executable `app.jar`.

Draw special attention to the `ENV` instruction. It's generally a good idea to set the `SPRING_PROFILES_ACTIVE` environment variable in any container image that contains a Spring Boot application. This makes it possible to configure beans and configuration properties that are unique to applications running in Docker.

In the case of the ingredient service, you're going to need a way to link the application to a Mongo database running in a separate container. By default, Spring Data attempts to connect to a Mongo database listening at port 27017 on localhost. But that was only the case when running everything locally and not in any containers. You'll need to configure the `spring.data.mongodb.host` property to tell Spring Data the hostname where Mongo will be available.

Although you may not yet know where the Mongo database will be running, you can configure Spring Data to connect to Mongo on a host named `mongo` when the `docker` profile is active by adding the following Docker-specific configuration to the `application.yml` file:

```
---
spring:
  profiles: docker

  data:
    mongodb:
      host: mongo
```

In a moment, when you fire up the Docker container, you'll map the `mongo` host to a Mongo database running in a different container. But now you're ready to build the Docker image. Using the Maven wrapper, execute the `package` and `dockerfile:build` goals to build the JAR file, and then build the Docker image:

```
$ mvnw package dockerfile:build
```

At this point, you can verify that the image is in your local image repository by using the `docker images` command (the `CREATED` and `SIZE` columns were omitted for easier readability and to fit within the margins of this page):

```
$ docker images
REPOSITORY                                TAG                IMAGE ID
tacocloud/ingredient-service             latest            7e8ed20e768e
```

Before you can start the container, you'll need to start a container for the Mongo database. The following command line runs a new Docker container named `tacocloud-mongo` with a Mongo 3.7.9 database:

```
$ docker run --name tacocloud-mongo -d mongo:3.7.9-xenial
```

Now, you can finally run the ingredient service container, linking it to the Mongo container you just started:

```
$ docker run -p 8080:8081 \
  --link tacocloud-mongo:mongo \
  tacocloud/ingredient-service
```

The `docker run` command shown here has several important components worth noting:

- Because you've configured the Spring Boot application in the container to run on port 8081, the `-p` parameter maps the internal port to the host's port 8080.
- The `--link` parameter links your container to the container named `tacocloud-mongo` and assigns it a hostname of `mongo` so that Spring Data can connect to it with that hostname.
- Finally, you specify the name of the image (in this case, `tacocloud/ingredient-service`) to run in a new container.

Now that you have a Docker image built and have proven it to run as a local container, you can take it to the next level by pushing the image to Dockerhub or some other Docker image repository. If you have an account on Dockerhub and are logged in, then you can push the image using Maven like this:

```
$ mvnw dockerfile:push
```

From that point, you can deploy the image to almost any environment that supports Docker containers, including AWS, Microsoft Azure, and Google Cloud Platform. Pick your environment and follow the platform-specific instructions for deploying Docker containers. Here are links to instructions for a few popular cloud platforms:

- *AWS*—<https://aws.amazon.com/getting-started/tutorials/deploy-docker-containers/>
- *Microsoft Azure*—<https://docs.docker.com/docker-for-azure/deploy/>
- *Google Cloud Platform*—<https://cloud.google.com/kubernetes-engine/docs/tutorials/hello-app>

- *Pivotal Web Services (PWS)*—<https://docs.run.pivotal.io/devguide/deploy-apps/push-docker.html>
- *Pivotal Container Service (PKS)*—<https://pivotal.io/platform/pivotal-container-service>

## 19.5 *The end is where we begin*

Over the past few hundred pages, we've gone from a simple start—or `start.spring.io`, more specifically—to deploying an application in the cloud. I hope that you've had as much fun working through these pages as I've had writing them.

But while this book must come to an end, your Spring adventure is just beginning. Using what you've learned in these pages, go build something amazing with Spring. I can't wait to see what you come up with!

### **Summary**

- Spring applications can be deployed in a number of different environments, including traditional application servers, platform-as-a-service (PaaS) environments like Cloud Foundry, or as Docker containers.
- When building a WAR file, you should include a class that subclasses `SpringBootServletInitializer` to ensure that Spring's `DispatcherServlet` is properly configured.
- Building as an executable JAR file allows a Spring Boot application to be deployed to several cloud platforms without the overhead of a WAR file.
- Containerizing Spring applications is as simple as using Spotify's Dockerfile plugin for Maven. It wraps an executable JAR file in a Docker container that can be deployed anywhere Docker containers can be deployed, including cloud providers such as Amazon Web Services, Microsoft Azure, Google Cloud Platform, Pivotal Web Services (PWS), and Pivotal Container Service (PKS).

# Spring IN ACTION Fifth Edition

Craig Walls

Free eBook  
See first page

Spring Framework makes life easier for Java developers. New features in Spring 5 bring its productivity-focused approach to microservices, reactive development, and other modern application designs. With Spring Boot now fully integrated, you can start even complex projects with minimal configuration code. And the upgraded WebFlux framework supports reactive apps right out of the box!

**Spring in Action, Fifth Edition** guides you through Spring's core features, explained in Craig Walls' famously clear style. You'll roll up your sleeves and build a secure database-backed web app step by step. Along the way, you'll explore reactive programming, microservices, service discovery, RESTful APIs, deployment, and expert best practices. Whether you're just discovering Spring or leveling up to Spring 5, this Manning classic is your ticket!

## What's Inside

- Building reactive applications
- Spring MVC for web apps and RESTful web services
- Securing applications with Spring Security
- Covers Spring 5.0

For intermediate Java developers.

**Craig Walls** is a principal software engineer at Pivotal, a popular author, an enthusiastic supporter of Spring Framework, and a frequent conference speaker.

“This new edition is a comprehensive update that strikes the balance between practical instruction and comprehensive theory.”

—Daniel Vaughan  
European Bioinformatics Institute

“The go-to book for learning the Spring Framework and an excellent reference guide.”

—Colin Joyce, Cisco

“Everything you need to know about Spring and how to build cloud-native applications.”

—David Witherspoon, Parsons

“This book is the Spring developer's Swiss Army knife!”

—Riccardo Noviello  
Nuvio Software Solutions

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit  
[manning.com/books/spring-in-action-fifth-edition](http://manning.com/books/spring-in-action-fifth-edition)

Over 100,000 copies sold!

 MANNING

\$49.99 / Can \$65.99 [INCLUDING eBook]

ISBN-13: 978-1-61729-494-5  
ISBN-10: 1-61729-494-2



9 781617 294945